

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

E 496 1414460

**Media Processing Methods, Systems and Application  
Program Interfaces**

Inventor(s):  
Sohail Mohammed  
Kirt Debique  
Geoffrey Dunbar  
Patrick Nelson  
Rebecca Weiss  
Sumedh Barde  
Adil Sherwani  
Robin Speed  
Alexandre Grigorovitch

ATTORNEY'S DOCKET NO. ms1-1873us

## **TECHNICAL FIELD**

This invention relates to media processing methods, systems and application program interfaces.

## **BACKGROUND**

As multimedia systems and architectures evolve, there is a continuing need for systems and architectures that are flexible in terms of implementation and the various environments in which such systems and architectures can be employed. As an example, consider the following as flexibility is viewed from the vantage point of software applications that execute in conjunction with such multimedia systems.

When it comes to rendering multimedia presentations, some software applications are very basic in terms of their functionality. That is, these basic types of applications might simply wish to provide a multimedia system with only a small amount of data that pertains to the presentation and have the multimedia system itself do the remainder of the work to render the presentation. Yet other more complex types of application may wish to be more intimately involved with the detailed processing that takes place within the multimedia system.

Against this backdrop, there is a continuing need to provide multimedia systems and architectures that meet the needs of applications that are distributed along a spectrum of simple applications to complex applications.

## **SUMMARY**

Media processing methods, systems and application program interfaces (APIs) are described. In but one embodiment, a media engine component, also

1 referred to as a media engine, provides a simple and unified way of rendering  
2 media from an origin to a destination of choice without requiring intimate  
3 knowledge about the underlying components, their connectivity and management.  
4 Clients of the media engine need not worry about how to render the particular  
5 media, but rather can simply focus on what media to render and where to render  
6 the media. In at least one embodiment, a media session is provided and is used by  
7 the media engine and provides a mechanism by which additional components are  
8 made transparent to the application and, in at least some embodiment, the media  
9 engine.

10 In some embodiments, the media engine and media session provide a  
11 simple API for building, configuring, and manipulating a pipeline of components  
12 (e.g. media sources, transforms, and sinks) for media flow control between an  
13 origin and one or more destinations.

## 14 15 **BRIEF DESCRIPTION OF THE DRAWINGS**

16 Fig. 1 is a block diagram of a system in accordance with one embodiment.

17 Fig. 2 is a block diagram of a system in accordance with another  
18 embodiment.

19 Fig. 3 is a flow diagram that describes steps in a method in accordance with  
20 one embodiment.

21 Fig. 4 is a block diagram of a system in accordance with another  
22 embodiment.

## DETAILED DESCRIPTION

### Overview

Fig. 1 illustrates a high level block diagram of an exemplary system in accordance with one embodiment, generally at 100. In one or more embodiments, system 100 is implemented in software. In system 100, an application 102 interacts with a media engine component or more simply a media engine 104. In at least some embodiments, media engine 104 serves as a central focal point of an application that desires to somehow participate in a *presentation*. A presentation, as used in this document, refers to or describes the handling of media content. In the illustrated and described embodiment, a presentation is used to describe the format of the data that the media engine is to perform an operation on. Thus, a presentation can result in visually and/or audibly presenting media content, such as a multimedia presentation in which both audio and accompanying video is presented to user via a window executing on a display device such as a display associated with a desk top device. A presentation can also result in writing media content to a computer-readable medium such as a disk file. Thus, a presentation is not simply limited to scenarios in which multimedia content is rendered on a computing device. In some embodiments, operations such as decoding, encoding and various transforms (such as transitions, effects and the like), can take place as a result of the presentation.

In one embodiment, media engine 104 exposes particular application program interfaces that can be called by application 102. One implementation of application program interfaces that are exposed by media engine 104 is described in the section entitled "Application Program Interfaces" below.

1 In accordance with the illustrated and described embodiment, the media  
2 engine 104 can use several components among which include a media session 106,  
3 one or more media sources 108, one or more transforms 110 and one or more  
4 media sinks 112, 114. One advantage of various illustrated and described  
5 embodiments is that the described system is a pluggable model in the sense that a  
6 variety of different kinds of components can be utilized in connection with the  
7 systems described herein. Also comprising a part of system 100 is a destination  
8 116, which is discussed in more detail below. In at least one embodiment,  
9 however, a destination is an object that defines where a presentation is to be  
10 presented (e.g. a window, disk file, and the like) and what happens to the  
11 presentation. That is, a destination can provide sink objects into which data flows.

12 In at least one embodiment, media session 106 can use media sources,  
13 transforms and media sinks. One reason for providing a media session 106 is to  
14 abstract away the specific details of the existence of and interactions between  
15 media sources, transforms and media sinks from the media engine 104 and the  
16 application 102. That is, in some embodiments, the components that are seen to  
17 reside inside the media session 106 are not visible, in a programmatic sense, to the  
18 media engine 104. This permits the media engine 104 to execute so-called "black  
19 box" sessions. That is, the media engine 104 can interact with the media session  
20 106 by providing the media session certain data, such as information associated  
21 with the media content (e.g. a URL) and a destination 116, and can forward an  
22 application's commands (e.g. open, start, stop and the like) to the media session.  
23 The media session 106 then takes the provided information and creates an  
24 appropriate presentation using the appropriate destination.  
25

1 A media source 108 comprises a component that knows how to read a  
2 particular type of media content from a particular source. For example, one type  
3 of media source might capture video from the outside world (a camera), and  
4 another might capture audio (a microphone). Alternately or additionally, a media  
5 source might read a compressed data stream from disk and separate the data  
6 stream into its compressed video and compressed audio component. Yet another  
7 media source might get such data from the network.

8 Transforms 110 can comprise any suitable data handling components that  
9 are typically used in presentations. Such components can include those that  
10 uncompress compressed data and/or operate on data in some way, such as by  
11 imparting an effect to the data, as will be appreciated by the skilled artisan. For  
12 example, for video data, transforms can include those that affect brightness, color  
13 conversion, and resizing. For audio data, transforms can include those that affect  
14 reverberation and resampling. Additionally, decoding and encoding can be  
15 considered as transforms.

16 Media sinks 112 and 114 are typically associated with a particular type of  
17 media content. Thus, audio content might have an associated audio sink such as  
18 an audio renderer. Likewise, video content might have an associated video sink  
19 such as a video renderer. Additional media sinks can send data to such things as  
20 computer-readable media, e.g. a disk file and the like.

21 In one embodiment, the media engine 104 provides various functionalities  
22 that can be utilized directly or indirectly by application 102. For example, media  
23 engine 104 can provide support for linear or simple (e.g. asf, mp3, etc.) and non-  
24 linear or composite (e.g. AAF, ASX, M3U, etc.) media sources. Alternately or  
25 additionally, the media engine 104 can provide transport control for media content

(e.g. play, pause, stop, rewind, forward, rate, scrubbing). Alternately or additionally, the media engine 104 can provide asynchronous building and management of a media pipeline given a source of media content. Alternately or additionally, the media engine 104 can provide for automatic resolution of media sources given a URL. Alternately or additionally, the media engine 104 can provide for automatic resolution of decoders, encoders, color converters, etc. to fully specify the connections between a source and a sink (e.g. format negotiation). Alternately or additionally, the media engine 104 can provide access to individual components in the media pipeline for configuration. Alternately or additionally, the media engine 104 can provide support for applying transformations, e.g. effects such as equalization, brightness control, and the like to media streams. Alternately or additionally, the media engine 104 can provide for quality of service notifications that enable clients to adjust pipeline configuration in order to achieve desired performance and user experience. Alternately or additionally, the media engine 104 can provide a service provider model to enable support for controlling media pipelines that cannot be factored into sources, transforms, and sinks. For example, a client of the media engine 104 can be able to control the flow of media samples in a distributed scenario across machine boundaries. Alternately or additionally, the media engine 104 can provide for synchronization of audio/video and other streams. Alternately or additionally, the media engine 104 can make provisions for processing metadata associated with particular media content. Such will become more apparent as the description below is read.

## Stream Selection

Stream selection, as used in this document, refers to the process whereby the media engine 104 chooses which media streams, from the underlying media source to process data from. In the embodiment described in this document, the media engine 104 supports multiple different modes of stream selection.

The first mode is referred to as *automatic stream selection*, in which the media engine 104 is responsible for selecting which streams are used. Automatic stream selection is the default stream selection mode. A second mode is referred to as *manual stream selection*, in which the application has control over exactly which streams are used. A *stream selector service*, which is exposed by the media engine, can be used to control advanced stream selection functionality.

### Automatic Stream Selection

In the illustrated and described embodiment, the default stream selection mode for the media engine is automatic stream selection in which the media engine selects the stream to use (with the assistance of the media source). The stream selection criteria, expressed as a property store, are passed to the media engine as part of the configuration parameters to the *openXXX* methods described below. The default, expressed by not putting any stream selection criteria into the configuration property store, will cause the media engine 104 to simply make the default selection of streams. The stream selection criteria are specified in the property store passed into the media engine 104 as part of the *openXXX* methods. The criteria can be changed on the fly using the stream selector service, using the `IMFStreamSelector::SetAutomaticStreamSelectionCriteria` method, which is



1 described in more detail in the section entitled "Application Program Interfaces"  
2 below.

3 With respect to interleaved streams, consider the following. If the stream  
4 selection criteria allow the type MFMediaType\_Interleaved, then any interleaved  
5 streams will be selected and exposed to the destination. In the case where the  
6 MFMediaType\_Interleaved is not selected by the criteria, the media engine will  
7 automatically insert a demultiplexer for any interleaved streams, and then perform  
8 stream selection on the demultiplexed streams as per the stream selection criteria.

### 9 10 Manual Stream Selection

11 Manual stream selection occurs when the application takes complete  
12 control of the stream selection process. Generally, this implies that the application  
13 has some specific knowledge of the media source in use, and can determine the  
14 relationships between streams through some source-specific knowledge.

15 Manual stream selection is enabled through the stream selector service by  
16 using the IMFStreamSelector::SetManualStreamSelection method (described  
17 below) to enable it. Note that, in one embodiment, manual stream selection can  
18 only be enabled or disabled while the media engine 104 is in the stOpened or  
19 stInitial states (i.e. basically, when the media engine is not running).

20 To implement manual stream selection, the application should set itself as a  
21 delegate for the MENewPresentation event. During processing of this event the  
22 application should configure the presentation descriptor (the value of the  
23 MENewPresentation event) with the desired stream selection. The media engine  
24 will inspect this presentation for the desired stream selection when the event  
25 delegate completes.

## Opening and Closing Media Content

In at least one embodiment, media engine 104 provides a variety of “*open*” methods or calls that can be utilized by an application to specify a data source in a variety of manners, and a destination for the data. The various *open* calls provide different mechanisms for specifying the source of the data. In this embodiment, media engine 104 is designed to be the common media control interface for the application. Calling an *open* method results in the media engine building a data flow pipeline for processing the data. This process can involve a negotiation between data source, stream selection criteria, and the destination. Once this process is complete and the pipeline is fully configured, an event can be sent to the application. In the implementation example given below, this event is referred to as the MEMediaOpened event. This and other events are more fully described in a section entitled “Events” below.

In one embodiment, media engine 104 exposes application program interfaces for supporting one or more functionalities described just below. For a specific implementation example, the reader is referred to the section entitled “Application Program Interfaces” below.

A first functionality that can be provided by the media engine’s APIs is a functionality that causes the media engine 104 to use a source resolver to create a media source that is able to read the media content at a specified URL. In the implementation example given below, an exemplary method IMFMediaEngine::OpenURL() is described.

A second functionality that can be provided by the media engine’s APIs is a functionality as follows. If an application has already created a media source and

1 would like the presentation to use that media source, the application can call a  
2 method that causes the media engine 104 to use the source that the application has  
3 created. In the implementation example given below, an exemplary method  
4 IMFMediaEngine::OpenSource() is described. In this particular instance, an  
5 application can be more involved in the presentation process insofar as being able  
6 to create and utilize its own media source.

7 A third functionality that can be provided by the media engine's APIs is a  
8 functionality that can be utilized by applications that do not have a media source,  
9 but instead have an object that implements an interface from which a media source  
10 object can be obtained. In the implementation example given below, an  
11 exemplary method IMFMediaEngine::OpenActivate() is described.

12 A fourth functionality that can be provided by the media engine's APIs is a  
13 functionality for applications that have an object from which sequential data can  
14 be obtained. In the implementation example described below, an exemplary  
15 method IMFMediaEngine::OpenByteStream() is described.

16 A fifth functionality that can be provided by the media engine's APIs is a  
17 functionality that can be utilized by advanced or complex applications.  
18 Specifically, some advanced applications may wish to create a partial topology  
19 defining the sources, sinks, and transforms to be used in the presentation, instead  
20 of relying on the media engine and media session to do it. In the illustrated and  
21 described embodiment, these applications can supply the partial topology and can  
22 by-pass any source-resolution that would otherwise be done. In the  
23 implementation example described below, an exemplary method  
24 IMFMediaEngine::OpenTopology() is described.

1 A sixth functionality that can be provided by the media engine's APIs is a  
2 functionality that closes the current media content. The media engine 104 can  
3 subsequently be reused for new media by using any of the above "Open" calls. In  
4 the implementation example described below, an exemplary method  
5 IMFMediaEngine:Close() method is described.

### 6 7 **Media Engine Presentation Control**

8 In accordance with one embodiment, media engine 104 provides  
9 functionality for a number of different presentation control features among which  
10 include functionalities that start a presentation, stop a presentation, pause a  
11 presentation, and start at a certain time, each of which is described under its own  
12 heading below.

#### 13 14 **Start**

15 In accordance with one embodiment, a *start* method is provided and can be  
16 called by an application to start a presentation. The *start* method utilizes different  
17 parameters to facilitate starting a presentation. In accordance with one  
18 embodiment, the *start* method provides an asynchronous way to start processing  
19 media samples. This operation gets the media samples moving through the  
20 pipeline and delivered to the appropriate sinks. A *StartOffset* parameter specifies a  
21 media location at which to start processing. A *Duration* parameter specifies the  
22 length of media that is to be processed beginning at the *StartOffset*. In some  
23 embodiments, instead of a *Duration* parameter, an *EndOffset* parameter can be  
24 used to specify a media location at which to end processing. In the event that  
25 these parameters are not used, default parameters are utilized that request a

1 presentation to start at the beginning of the data and play until the end of the data.  
2 An event (described below as a MEMediaStarted event) can also be generated and  
3 sent to the application to mark the completion of this asynchronous operation. The  
4 *start* method can also be called while the media engine 104 is in the course of a  
5 presentation. In this case, the *start* method seeks to a new location in the media  
6 content and starts the presentation from the new location. An exemplary start  
7 method IMFMediaEngine::Start() is described below.

### 8 9 Stop

10 In accordance with one embodiment, a *stop* method is provided and can be  
11 called by an application to stop the presentation. The *stop* method provides an  
12 asynchronous way to stop media sample processing in the media engine.  
13 Responsive to calling the *stop* method, an event can be generated and sent to the  
14 application to notify the application of the completion of this operation. An  
15 exemplary *stop* method IMFMediaEngine::Stop() is described below.

### 16 17 Pause

18 In accordance with one embodiment, a *pause* method is provided and can  
19 be called by an application to pause the presentation. The *pause* method provides  
20 an asynchronous way to stop media processing in the media engine. Responsive  
21 to calling the *pause* method, an event can be generated and sent to the application  
22 to notify the application of the completion of this operation.

### Start at a Certain Time

In accordance with one embodiment, a *start at time* method is provided and can be called by an application to start a presentation at a specified time. Specifically, this method causes the media engine 104 to start a presentation at a certain time. The “start at time” parameter is given with respect to an external clock. By default, the external clock is the system clock; however, the external clock may also be an external clock that is provided by the application.

### Information Querying About the Presentation

In accordance with one embodiment, a set of APIs are provided that enable applications to obtain information that pertains to the presentation, the media engine 104, and the media session 106. As an example, consider the following non-exclusive methods that can be called by the application.

#### GetCapabilities

In one embodiment, a method is provided that enables the application to be exposed to various capabilities of the media engine and media session. Some of the capabilities that the application can ascertain can be capabilities that can change during the course of a presentation. As an example, various capabilities can include, without limitation, “can start”, “can skip forward”, “can skip backward”, “can skip TL node”, “can seek”, and “can pause”. An example method IMFMediaEngine::GetCapabilities is described below.

### GetMetadata

In one embodiment, a method is provided that enables the application to obtain a property store which can be queried for various metadata associated with or concerning the presentation. Examples of such metadata include, without limitation, duration, title, author and the like. An exemplary method IMFMediaEngine::GetMetadata() is described below.

### GetDestination

In one embodiment, a method is provided that enables the application to ascertain the current destination. This method returns a pointer to the destination in use. An exemplary method IMFMediaEngine::GetDestination() is described below.

### GetStatistics

In accordance with one embodiment, a method is provided that enables the application to ascertain statistics associated with a media engine. Using this method, an application can query statistics of the current configuration of the media engine. These statistics may be generated by the media engine, or may be aggregated from the underlying components (media sources, transforms, media sinks). An exemplary method IMFMediaEngine::GetStatistics() is described below.

### GetState

In accordance with one embodiment, a method is provided that enables the application to ascertain the current state of the media engine (e.g. started, stopped,

1 paused, etc.). An exemplary method IMFMediaEngine::GetState() is described  
2 below.

### 3 4 GetClock

5 In accordance with one embodiment, a method is provided that enables the  
6 application to retrieve the clock according to which the media engine is  
7 presenting. The application can use this information to monitor the progress of the  
8 presentation. Monitoring the progress of the presentation can enable the  
9 application to display a counter that indicates the current location of the  
10 presentation. An exemplary method IMFMediaEngine::GetClock() is described  
11 below.

### 12 13 ShutDown

14 In accordance with one embodiment, a method is provided that causes all  
15 resources that are used by the media engine to be properly shut down and released.  
16 The method also releases all references to other components (including media  
17 sessions), shutting them down as well. An exemplary method  
18 IMFMediaEngine::Shutdown() is described below.

### 19 20 Events

21 In accordance with one embodiment, the media engine 104 supports an  
22 interface for generating events that mark the completion of asynchronous methods  
23 invoked on the media engine. The media engine 104 can also notify the client (i.e.  
24 application) about unsolicited events that occur in the media pipeline through the  
25 same mechanism. For example, the media engine can generate events that indicate



1 changes in the presentation and quality control notifications. In addition, it is to be  
2 appreciated and understood that events can also be generated by the media session  
3 106 and/or components inside the media session for propagation to the application.

4 As a non-exclusive example of such events, consider the following  
5 discussion which highlights certain events under their own separate headings.

#### 6 7 MENewPresentation

8 This event is accompanied by a presentation descriptor that describes how  
9 the presentation coming out of the media source(s) will look. As an example, in  
10 some embodiments, up until this time, the application does not know what type of  
11 content comprises the presentation, e.g. audio, video, multiple streams, and the  
12 like. The presentation descriptor allows the application to ascertain this  
13 information. In one embodiment, this event will be received once after one of the  
14 *open* calls described above is made, and once for every new presentation that the  
15 media source will output subsequently in timeline scenarios. Applications can  
16 optionally use this information to configure the destination in anticipation of this  
17 new presentation.

18 In one embodiment, the MENewPresentation event is fired at topology  
19 creation time, and indicates that a new presentation is being configured, for which  
20 a destination method entitled IMFDestination::GetOutputInfo method, will get  
21 called for every stream in the new presentation. The application may set itself as a  
22 delegate for this event on the media engine, in which case the destination is  
23 guaranteed to not get invoked until the delegate has finished processing the event  
24 and called the callback function.

1 With respect to the *value* of this event, consider the following. The value  
2 contains a pointer to an IMFPresentationDescriptor object, which will have stream  
3 descriptors for the output streams of the presentation, with partially-specified  
4 media types. These stream descriptors will be the same IMFStreamDescriptor  
5 pointers that will be sent to the GetOutputInfo calls on the destination. The  
6 metadata property store object of this presentation descriptor will have a pointer to  
7 the input presentation descriptor, saved under the property key  
8 MFPKEY\_ENGINE\_SourcePresentation, containing additional information about  
9 the input streams from the media source.

#### 11 MEMediaOpened

12 The MEMediaOpened event indicates the completion of the *openXXX*  
13 asynchronous calls on the media engine 104. In some embodiments, when this  
14 event is fired, the first presentation for the media has been configured and fully  
15 loaded, and is ready to start. The media engine can accept *start* calls before this  
16 event is fired. In practice, however, the execution of the *start* calls will typically  
17 be delayed until the *open* has completed. In one embodiment, unless the *open* call  
18 failed, MEMediaOpened is guaranteed to be preceded by a MENewPresentation  
19 event, for configuring the presentation before it can be ready to start. With respect  
20 to the status of the event, the status is designated as "OK" if the *open* call  
21 succeeded, and a failure code if the *open* call failed. With respect to the *value* of  
22 this event, if the *open* call succeeded, the data object contains a pointer to an  
23 IMFPresentationDescriptor object, with fully-specified media types for the output  
24 streams of the presentation. This object will include any additional output streams  
25 that would have been created if the media engine received a collection of

IMFOutputInfos from the destination for a particular output stream. The metadata property store object of this presentation descriptor will have a pointer to the input IMFPresentationDescriptor object, saved under the property key MFKEY\_ENGINE\_SourcePresentation, containing additional information about the input streams from the media source. This input IMFPresentationDescriptor object will be the same one that was present in the property store of the data object for the corresponding MENewPresentation event.

#### MEMediaStarted

The MEMediaStarted event indicates the completion of the asynchronous operation begun by calling *start* on the media engine. With respect to the status of the event, the status is “OK” if the *start* call succeeded, and a failure code if the *start* call failed. With respect to the *value* of the event, if the *start* call succeeded, the data value propvariant contains the start time in 100ns units.

#### MEMediaStopped

The MEMediaStopped event indicates the completion of the asynchronous operation begun by calling *stop* on the media engine. With respect to the status of the event, the status is “OK” if the *stop* call succeeded, and a failure code if the *stop* failed.

#### MEMediaPaused

The MEMediaPause event indicates the completion of the asynchronous operation begun by calling *pause* on the media engine. With respect to the status

1 of the event, the status is “OK” if the *pause* call succeeded, and a failure code if  
2 the *pause* call failed.

#### 3 4 MEMediaEnded

5 The MEMediaEnded event indicates that the last sample of data from the  
6 active media source has been rendered.

#### 7 8 MEMediaClosed

9 The MEMediaClosed event indicates the completion of the asynchronous  
10 operation begun by calling *close* on the media engine. At this point, other *open*  
11 calls can be executed. With respect to the status of the event, the status is “OK” if  
12 the *close* call succeeded, and a failure code if the *close* failed.

#### 13 14 MEPresentationSwitch

15 The MEPresentationSwitch event is fired when a switch is made from one  
16 presentation to the next, and indicates that the active running presentation has been  
17 replaced with a new presentation. In one embodiment, this event is fired when the  
18 next presentation is actually executed or played. Each MEPresentationSwitch  
19 event is guaranteed to be preceded by a corresponding MENewPresentation event,  
20 for configuring the presentation before it can be switched to.

21 With respect to the *value* of the event, the *value* contains a pointer to the  
22 presentation descriptor of the new presentation that has been switched to. This  
23 object is similar to that given out in the MEMediaOpened event, as it has fully-  
24 specified media types on the output nodes and contains a pointer to the source  
25

1 presentation descriptor in its metadata property store object, under the property  
2 key MFPKEY\_ENGINE\_SourcePresentation.

### 3 4 MEOutputsUpdated

5 The MEOutputsUpdated event indicates that the media engine has received  
6 a MEDestinationChanged event from the destination, and has reconfigured the  
7 current presentation to handle the changes from the destination. The  
8 MEOutputsUpdated event is not preceded by a MENewPresentation event, but  
9 every MEOutputsUpdated event corresponds to a MEDestinationChanged event  
10 that the Engine receives from the destination.

11 With respect to the *value* of this event, the data object contains a pointer to  
12 the presentation descriptor of the current presentation after being reconfigured  
13 with the destination changes. This object is similar to that given out in the  
14 MEMediaOpened and MEPresentationSwitch events, as it has fully-specified  
15 media types on the output nodes and contains a pointer to the source presentation  
16 descriptor in its metadata property store object, under the property key  
17 MFPKEY\_ENGINE\_SourcePresentation.

### 18 19 MEEngineStateChanged

20 The MEEngineStateChanged event indicates that the state of the media  
21 engine has changed. The “value” member of this event is an  
22 IMFMediaEngineState pointer, from which the application can get more  
23 information about the state. The IMFMediaEngineState interface is described  
24 further below. Note that because of the asynchronous nature of the various events,  
25 the engine state may have changed yet again by the time the application sees this

1 event. This event (and its value) is useful when an application wants to know  
2 about every single state change in the order it occurs. On the other hand if the  
3 application only wants to know the latest state, then it should use the GetState()  
4 method instead to synchronously fetch the current state of the engine.

### 6 MECapabilitiesChanged

7 The MECapabilitiesChanged event is fired by the engine when the set of  
8 allowed operations on the engine changes. The media engine also bubbles this  
9 event up from underlying components. This event and its arguments are defined  
10 in more detail in the section titled "Engine Capabilities" below.

### 12 MEMediaRateChanged

13 When the application has changed the playback rate, this event is sent to  
14 indicate that playback is now happening at the new rate. The value is a floating-  
15 point number indicating the new rate. The status code indicates whether the rate  
16 change occurred successfully.

### 18 Engine Capabilities

19 In accordance with one embodiment, not all operations on the media engine  
20 are allowed at all times. For example, certain operations may be disabled when  
21 the media engine or components under it are in a state where that operation does  
22 not make sense, e.g. *start* will be disabled until *open* completes. Additionally,  
23 certain operations may also be disabled based on the media content, e.g. an ASX  
24 may allow skipping or seeking during certain clips.

Applications typically need to know what operations are available at any particular moment, both at the media engine level and on the other interfaces that the applications have access to. Applications also typically need to know when the set of allowed operations changes. One reason that applications need this information is so that the applications can enable or disable corresponding user interface (UI) features.

In the presently-described embodiment, these needs are satisfied by the GetCapabilities method on IMFMediaEngine and the MECapabilitiesChanged event.

#### **IMFMediaEngine::GetCapabilities**

This method returns a DWORD which is a bitwise OR of the following bits.

Bit	Description
<i>MFENGINECAP_START</i>	This bit is set if the media engine currently allows Start() with a start time of PRESENTATION_CURRENT_POSITION. An application will typically enable the Play button based on this.
<i>MFENGINECAP_SKIPFORWARD</i>	This bit is set if the media engine currently allows Start() with a time format of GUID_ENTRY_RELATIVE and a positive time value. In the case of timeline sources this means skipping to the next node; for non-timeline sources this typically means skipping to the next segment as defined by the source. An application will typically enable the SkipForward button based on this.

<i>MFENGINECAP_SKIPBACKWARD</i>	This bit is set if the media engine currently allows Start() with a time format of GUID_ENTRY_RELATIVE and a negative time value. An application will typically enable the SkipBackward button based on this.
<i>MFENGINECAP_SKIPTLNODE</i>	This bit is set if the media engine allows calling Start() with the GUID_TIMELINE_NODE_OFFSET time format. An application will typically enable its playlist UI based on this.
<i>MFENGINECAP_SEEK</i>	This bit is set if the engine allows calling Start() with any value other than the ones called out above. An application will typically enable its seek bar based on this.
<i>MFENGINECAP_PAUSE</i>	This bit is set if the engine allows calling Pause(). An application will typically enable the Pause button based on this.

As may be gleaned from the above table, each capability bit corresponds to a method on the media engine. If the bit is set then there is a *good chance* that method will succeed. However due to the dynamic nature of the engine, this cannot be guaranteed.

In one embodiment, there are no capability bits corresponding to Stop(), Close(), or Shutdown(), as these operations are always allowed.

### **MECapabilitiesChanged**

This event is fired when the set of allowed operations on the media engine changes. This event is also bubbled up by the engine from components underneath it. GetValue on this event returns a PROPVARIANT whose type is VT\_UNKNOWN and punkVal member contains a pointer to IMFCapChange.



This interface contains more information about what capabilities changed and on which component. The section just below describes this interface in detail. Preliminarily, the description below sets forth a way that enables capability changes from components underlying the media engine to be sent to the application. The approach described below alleviates a need for the media engine to wrap every single capability of every single underlying component.

### **IMFCapChange**

interface IMFCapChange : IUnknown

```
{  
    HRESULT GetOriginatorIID( [out] IID *piid );  
    HRESULT GetOriginatorPointer( [out] IUnknown **ppunk );  
    HRESULT GetNewCaps( [out] DWORD *pdwNewCaps );  
    HRESULT GetCapsDelta( [out] DWORD *pdwCapsDelta );  
}
```

### **IMFCapChange::GetOriginatorIID**

GetOriginatorIID returns the IID for the interface whose capabilities changed. In the case of the media engine the only valid value is IID\_IMFMediaEngine. In the case of components that expose multiple interfaces and/or services, the IID indicates how to interpret the bits obtained by GetNewCaps and GetCapsDelta.

### **Syntax**

HRESULT GetOriginatorIID( [out] IID \*piid );

### **Parameters**

*piid*

[out] IID of the interface whose capabilities changed.

### **Return Values**

1 If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### 2 3 IMFCapChange::GetOriginatorPointer

4 GetOriginatorPointer returns the pointer to the interface whose capabilities  
5 changed. If the IID returned by the previous method is IID\_IMFMediaEngine  
6 then this pointer does not add much value. In the case of the rate control service  
7 however, the application can use this pointer to query more detailed information  
8 about the new rate capabilities using IMFRate-specific methods.

#### 9 **Syntax**

10 HRESULT GetOriginatorPointer( [out] IUnknown \*\*punkOriginator );

#### 11 **Parameters**

12 *punkOriginator*

13 [out] Pointer to the interface whose capabilities changed.

#### 14 **Return Values**

15 If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### 16 17 IMFCapChange::GetNewCaps

18 GetNewCaps returns a DWORD containing a bitwise OR of the new  
19 capabilities. An application must interpret these bits differently based on the IID  
20 returned by GetOriginatorIID. In many cases this DWORD may be good enough  
21 for the application to update its UI; in some cases (such as IMFRate) the  
22 application may need to get more information via the pointer returned by  
23 GetOriginatorPointer.

#### 24 **Syntax**

25 HRESULT GetNewCaps( [out] DWORD \*pdwNewCaps );

## Parameters

*pdwNewCaps*

[out] Bitwise OR of the new capabilities.

## Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## IMFCapChange::GetCapsDelta

GetCapsDelta returns a DWORD which is a bitwise OR of the capabilities that just changed. This lets the application optimize what it does on a cap change. In the case of simple boolean caps (e.g. can pause or cannot pause), this is just an XOR of the previous value and the new value. However for more complex caps this bit may be set even though the previous value of the caps and the new value both have the bit set. For example, in the case of rate, if the range of positive rates changes then both the previous bit and the new bit will be set yet the delta will also be set to 1.

## Syntax

```
HRESULT GetCapsDelta( [out] DWORD *pdwCapsDelta );
```

## Parameters

*pdwCapsDelta*

[out] Bitwise OR of the capabilities that changed.

## Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## Media Session

1 As noted above, in the illustrated and described embodiment, a media  
2 session 106 is utilized to run a presentation. In the embodiments described above,  
3 the media session 106 is used by the media engine 104. One of the reasons for the  
4 existence of the media session is to allow third parties, also referred to as  
5 “providers,” to plug into and utilize the infrastructure support provided by the  
6 media engine 104 without necessarily factoring their design into such components  
7 as media sources, media sinks, and transforms. This enables system 100 in Fig. 1  
8 to be utilized to implement so-called “black box” sessions in which the application  
9 and media engine need not necessarily be familiar with or knowledgeable of the  
10 specific components that reside inside of the media session.

11 In one embodiment, the media session 106 enables the media engine 104 to  
12 control the data flow between a variety of sources and sinks through a single  
13 interface. In the example described below in the section entitled “Media Session  
14 Application Program Interfaces”, this interface is referred to as the  
15 IMFMediaSession interface.

16 In at least one embodiment, a media session 106 manages dataflow from  
17 the media source to one or more sinks. This can and typically does include the  
18 transformation and synchronization of media content. The media engine 104 has  
19 the responsibility of selecting the appropriate session type based on the  
20 sources/sinks. For example, some scenarios that can require different types of  
21 media sessions include, without limitation, (1) local machine playback/encode  
22 (Media Processing Session), (2) distributed playback (Distributed Media Session),  
23 (3) macromedia flash playback (Flash Media Session), and (4) a protected media  
24 path session. A protected media path session is provided for presenting protected  
25 (e.g. rights-management) content. In one embodiment, the session can reside in a

1 separate process from the application. In another embodiment, the session can  
2 reside in the same process as the application and can create a separate protected  
3 media path process where the media samples get decoded and processed.

#### 4 5 Media Session Configuration

6 In accordance with one embodiment, APIs are provided that enable the  
7 media engine 104 to configure the media session 106 for a presentation.

8 A first method, referred to as SetTopology, provides an asynchronous way  
9 of initializing a full topology on the media session. The media engine 104 calls  
10 this method once it has a full topology for the upcoming presentation. In one  
11 embodiment, the media session 106 sets this on a component known as the media  
12 processor (described in more detail below), which sets up the pipeline to get data  
13 from the media source through all of the transforms specified in the topology. An  
14 exemplary method IMFMediaSession::SetTopology() is described below. In some  
15 embodiments, the SetTopology() method can be combined with a  
16 ResolveTopology() method so that the method can take a partial topology and  
17 resolve it internally.

18 A second method, referred to as SetPresentationClock, provides a way for  
19 the media session to ensure that all components (e.g. media sinks and some media  
20 sources) subscribe to receive notifications from the same clock which is used to  
21 control the presentation. An exemplary method  
22 IMFMediaSession::SetPresentationClock() is described below.

23 A third method, referred to as SetConfigurationPropertyStore, provides an  
24 extensible set of configuration parameters that can be passed to the media session.

### **Media Session Presentation Control**

In accordance with one embodiment, media session 106 provides functionality for a number of different presentation control features among which include functionalities that start a presentation, stop a presentation, and pause a presentation. These functionalities are similar to the ones discussed above with respect to the media engine. Hence, for the sake of brevity, the description associated with these methods is not repeated.

### **Information Querying About the Media Session**

In accordance with one embodiment, a set of APIs are provided that enable the media engine 104 to obtain information from the media session 106. As an example, consider the following non-exclusive methods that can be called by the media engine.

A first method, referred to as GetSessionGUID allows the media engine 104 to ascertain a globally unique identifier that is associated with a particular implementation of a media session. This allows the media engine to differentiate between various types of sessions (black box sessions vs non-black box sessions, for example).

A second method referred to as GetSessionCapabilities allows the media engine 104 to ascertain certain capabilities of the media session. In one embodiment, the bits returned through this method have the same value as the bits in the table appearing above-- only the "MFENGINECAP"\_ is replaced by "MFSESSIONCAPS"\_ in all cases

### **Media Session Shutdown**

1 In one embodiment, a method referred to as Shutdown causes all resources  
2 used by the media session to be properly shut down and released. An exemplary  
3 method IMFMediaSession::Shutdown() is described below.

#### 4 5 **Media Session Events**

6 In much the same way that the media engine generates events for the  
7 application, the media session supports event generation functionality in which  
8 events can be generated by the media session and received by the media engine.  
9 For some of the events, the media engine performs a translation in which the  
10 session-generated event is translated to one of the events that the media engine  
11 typically generates and subsequently forwards to the application. For some  
12 session-generated events, the media engine acts on the information contained  
13 therein and does not propagate the events to the application.

14 An exemplary non-limiting list of events that can be generated by the media  
15 session and received by the media engine include a MESSessionStarted which is  
16 generated when a session is started; a MESSessionStopped which is generated when  
17 the session is stopped; a MESSessionEnded which is generated when the session is  
18 ended; a MESSessionPaused event which is generated when the session is paused;  
19 and a MEMediaRateChanged event which is generated when the media rate is  
20 changed.

21 Of course, other events can be generated by the media session and received  
22 by the media engine.

#### 23 24 **Exemplary Presentation Flow Process**

25

1        Having considered the system of Fig. 1 and the explanation above, consider  
2        now system 200 in Fig. 2. In this embodiment, similar numerals in the form of  
3        “2XX” have been utilized, where appropriate, to depict like or similar  
4        components. So, for example, the system of Fig. 2 comprises an application 202,  
5        media engine 204, media session 206, media sinks 212 and 214, and a destination  
6        216. Notice in this embodiment that there are three new or additional  
7        components—a media processor 207 and bit pumps 211a, 211b.

8        In this embodiment, media processor 207 is a component that effectively  
9        wraps one or more media sources 208 and one or more transforms 210. That is, in  
10       some instances, it may be desirable to both read media content and perform some  
11       type of transform on the data. In this situation, the media processor is a  
12       component that can permit the read/transform functionality to be accomplished by  
13       a single component. As an example, consider two different situations—one in  
14       which the application wishes to have compressed data processed by the media  
15       engine, and one in which the application wishes to have decompressed data  
16       processed by the media engine. In the first instance, the application might simply  
17       cause a media source to be created that reads compressed data. In the second  
18       instance, the application might simply cause a media processor to be created that  
19       comprises a media source that reads compressed data and a transform that  
20       decompresses the compressed. In this instance, the data that emerges from the  
21       media processor is decompressed data. In the described embodiment, the media  
22       processor “looks” the same to the application as the media source. That is, in this  
23       embodiment, the media processor exposes a media source interface so that it  
24       effectively looks like a media source—albeit with added functionality in the form  
25       of transforms 210.



1 With respect to bit pumps 211a, 211b, consider the following. The bit  
2 pumps provide a component that is effective to acquire the data that is processed  
3 by the media processor 207. In this embodiment, the bit pumps are used to *pull*  
4 data from the media processor 207. This is different from processing systems that  
5 are effectively *push models* where data is pushed down the pipeline. Here, the bit  
6 pumps pull the data from the media processor 207 and push the data to a  
7 respective media sink 212, 214. In one embodiment, there is a one-to-one  
8 relationship between a bit pump and an associated stream sink. Specifically, in  
9 this embodiment, each media sink can have one or more stream sinks. For  
10 example, an ASF file sink can have multiple stream sinks for audio and video  
11 streams and there will be one bit pump per stream. One advantage of using bit  
12 pumps is that bit pumps can relieve the media sink components from having to  
13 include the logic that pulls the data from the media processor 207 or media stream.  
14 This, in turn, can lead to easing the design complexities and requirements of the  
15 individual media sinks. For example, in some embodiments, it may be desirable to  
16 pull data from a particular stream or media processor and push the data to two  
17 different media sinks, e.g. a renderer and a disk file. In this instance and in  
18 accordance with one embodiment, a topology is built to route data to two or more  
19 different outputs and the media processor 207 shuttles data along those paths.  
20 Once the data gets to an associated bit pump, the bit pump is utilized to provide  
21 the data to its associated sink.

22 Given the description of the system of Fig. 2, the discussion that follows  
23 provides a general overview of a typical multimedia scenario, along with a  
24 description of the roles that the media engine 204 and media session 206 play in  
25 driving the presentation. In the discussion that follows, each of the media engine

1 (and its role) and media session (and its role) are discussed in sections under their  
2 own respective headings—i.e. “Media Engine Work” and “Media Session Work”.

### 3 4 **Media Engine Work**

5 In accordance with one embodiment, the work that the media engine 204  
6 performs during a presentation can be categorized, generally, under a number of  
7 different headings which appear below. The categories of media engine work  
8 include source resolution, setting up the media session, partial topology resolution,  
9 topology resolution and activation, presentation control, new presentations, and  
10 output changes.

### 11 12 **Source Resolution**

13 Source resolution pertains to the process by which the media engine 204  
14 causes the appropriate media source to be created for the particular type of data  
15 that is to be read and subsequently processed by the system. Thus, this process  
16 obtains a media source from which the multimedia data can be read. This process  
17 is relevant when, for example, the OpenURL or OpenByteStream methods  
18 (discussed above and below) are called to open the multimedia. In either case, the  
19 media engine 204 passes the URL or the Byte Stream, respectively, to a  
20 component known as a source resolver. If the source resolver is given a URL,  
21 then it looks at the scheme of the URL (e.g. file://, http://, etc) to create a Byte  
22 Stream that will read from the specified location.

23 In both cases, the source resolver is able to look at the contents of the Byte  
24 Stream to determine the format of the bits (ASF, AVI, MPEG, etc) so that a media  
25

1 source can be instantiated that will understand that format. The other *Open*  
2 functions discussed above and below specify the media source directly.

### 3 4 Setting up the Media Session

5 During this process, the media engine asks the media source that is created  
6 for a presentation descriptor. In some embodiments, the presentation descriptor  
7 may specify that a custom media session is to be used. In many cases, however,  
8 custom media sessions may not be used in which case a default media session can  
9 be instantiated.

### 10 11 Partial Topology Resolution

12 During partial topology resolution, the media engine obtains a presentation  
13 descriptor from the media source and notifies the application of that particular  
14 presentation via the event `MENewPresentation` described above. If the application  
15 is interested in using that event to configure the destination 216, the media engine  
16 waits for the application to finish handling the event.

17 The media engine then negotiates with the application-provided destination  
18 and the destination can create one or more media sinks for the outputs of the  
19 presentation. In some embodiments, media sinks can have already been created  
20 and the destination simply hands them over to the media engine.

21 The media engine constructs a “partial topology”, in the sense that the  
22 media engine indicates the source media streams and the output stream sinks,  
23 without necessarily specifying the transforms that will be needed to get there.  
24 Thus, referring to the Fig. 2 illustration, at this point in the process, components  
25 207, 208, 212, 214 and 216 have either been created and/or are referenced.

## Topology Resolution and Activation

In accordance with one embodiment, during topology resolution and activation, the media engine 204 asks the media session 206 to resolve the partial topology into a fully specified topology. The media engine 204 then sets the new fully-specified topology on the media session, which gives it to the media processor 207. As an example, consider that the media source that is created is one that reads a compressed WMV file. The sinks, on the other hand, are not configured to handle compressed data. Thus, during the topology resolution, the media session ascertains which transforms are necessary to provide the compressed data from the WMV file to the sinks and creates the appropriate transforms which, in this case, might comprise a decompressor and possibly resizers, color converters, resamplers, and the like.

In another embodiment, resolution and activation can be combined into a single operation. Specifically, the media engine 204 can set a partial topology on the media session 206 and the media session itself can resolve the partial topology into a fully-specified topology which it then provides to the media processor 207.

## Presentation Control

With the media session having been set up and the topology resolved and activated, the application 202 now can control the progress of the presentation by calling *start*, *stop*, and *pause* methods on the media engine 204. The media engine, in turn, forwards these calls to the media session 206, which handles the calls.

### New Presentations

As noted above, the media engine is configured to handle new presentations from the media source. As an example, consider the following. If, for example, the media source 208 is a timeline media source, then the media source 208 will notify the media session 206 of the upcoming new presentation by means of a new presentation event, which will, in turn, forward the event to the media engine 204. The media engine 204 can then repeat the partial topology resolution and topology resolution and activation processes described above, using a descriptor for the new presentation. Once playback of that new presentation is about to commence, an event MEPresentationSwitch (described above) is sent to the application 202.

### Output Changes

As noted above, the media engine 204 is configured to handle output changes. As an example, consider the following. If the media engine 204 receives an event from an application-provided destination notifying it of a change on the output side, the media engine 204 can go through the partial topology resolution and topology resolution and activation processes described above, using a descriptor for the existing presentation. Once presentation using the new outputs is about to commence, an event MEOutputsUpdated (described above) can be sent to the application.

### Media Session Work

The discussion below describes work that the media session 206 accomplishes in accordance with one embodiment. The work that the media session performs in accordance with this embodiment can be categorized,

1 generally, under the following categories: full topology resolution, media  
2 processor creation, time source selection, presentation control, bit pumps, new  
3 presentations and output changes, time line processing and content protection,  
4 each of which is discussed under its own separate heading below.

### 5 6 Full Topology Resolution

7 In performing the topology resolution process described above, the media  
8 session 206 can utilize a component called a *topology loader*. The topology loader  
9 is used to determine which transforms, such as transforms 210 that are necessary  
10 or desirable to provide the data from the media source(s) 208 to the media sink(s)  
11 212, 214.

### 12 13 Media Processor Creation

14 The media session 206 is responsible for creating the media processor 207.  
15 That is, the media session 206 owns the media processor 207. When the topology  
16 is set on the media session 206, the media session, in turn, sets the topology on the  
17 media processor 207. The media processor 207 follows the data flow laid out by  
18 the topology to transform data from the media source(s) 208 to the particular  
19 formats that are needed by the media sinks 212, 214.

### 20 21 Time Source Selection

22 One of the functions that the media session can perform pertains to time  
23 source selection. Specifically, upon starting a presentation, the media session 206  
24 can make a determination as to which of the available time sources will be used to  
25 drive the presentation. Each component can then run its part of the presentation in

1 synchronization with the time from the time source ascertained by the media  
2 session. The time source is also used in the presentation clock (owned by the  
3 media engine but given to the media session) for the purposes of reporting  
4 progress of the presentation.

5 Media sinks, such as sinks 212, 214, may optionally offer a time source.  
6 Typically, the audio renderer (i.e. audio sink) can offer a time source, and the time  
7 on the time source will be dictated by the audio device on the particular machine  
8 on which the presentation is presented. It is to be appreciated, however, that other  
9 media sinks may do so as well. In addition, a particular media source, e.g. live  
10 media sources such as device capture and network sources, may also provide some  
11 concept of time. In one embodiment, the media session takes care of attempting to  
12 make the time source it chooses run at a similar rate to that of the live media  
13 source. In one embodiment, the media session 206 can decide which of the time  
14 sources is the "highest priority" time source, and this time source is used by the  
15 main presentation clock, to which all clock-aware components synchronize their  
16 presentations.

### 17 18 Presentation Control

19 As noted above, the media session 206 can receive method calls to Start,  
20 Stop, and Pause from the media engine 204. These calls typically correspond to  
21 the applications calls that are made on the media engine 204.

22 The media session 206 can control the presentation via a Presentation  
23 Clock that it receives from the media engine 204. Starting, stopping and/or  
24 pausing the Presentation Clock results in all media sinks 212, 214 receiving  
25 notifications thereof and reacting appropriately. The media session 206 starts,

1 stops, and/or pauses the media processor 207 by respectively calling its start, stop,  
2 and/or pause methods directly.

3 The media session 206 is configured, in this embodiment, to send an event  
4 to the media engine 204 after a given operation has been completed by all streams.

### 6 Bit Pumps

7 In accordance with the described embodiment, the media session 206 drives  
8 all data flow from the media processor 207 to the media sinks 212, 214. In  
9 accordance with one embodiment, bit pumps are employed as the abstraction that  
10 encapsulates the data flow management functionality within the media session. In  
11 the specific implementation example that is illustrated in Fig. 2, each media sink is  
12 associated with a bit pump. Specifically, media sink 212 is associated with bit  
13 pump 211a, and media sink 214 is associated with bit pump 211b. Additionally,  
14 in this specific example, the bit pumps operate by first requesting, from an  
15 associated media sink, a sample allocation. When the sample allocation request is  
16 filled by the media sink, the bit pump requests that the media processor 207 fill the  
17 sample with data. Once the sample is filled with data, the sample is handed by the  
18 bit pump to its associated media sink and the bit pump makes a request on the  
19 media sink for another sample allocation. It should be appreciated and understood  
20 that at any given time, more than one sample can be “in the air”. In one  
21 embodiment, the media session is responsible for determining how many samples  
22 are supposed to be outstanding at once and making the appropriate requests from  
23 the media sinks’ stream sinks.

### 25 New Presentations and Output Changes



1 In accordance with this embodiment, media session 206 is responsible for  
2 forwarding media processor 207's notification of an upcoming new presentation to  
3 media engine 204 and participating with topology resolution and activation, as  
4 described above in connection with the media engine.

#### 6 Time Line Processing

7 In accordance with one embodiment, media session 206 is configured to  
8 reduce glitches at presentation startup time and when transitioning between  
9 presentations in a timeline.

10 In accordance with this embodiment, at startup time, media session 206 will  
11 get the first few samples of media data from media processor 207 and deliver them  
12 to the media sinks 212, 214 before starting the clock associated with the  
13 presentation. This processes uses a special "prerolling" capability on the media  
14 sinks that allows the media sinks to receive data before actually being started. In  
15 this embodiment, it is only after the media sinks receive data via the pre-rolling  
16 capability that media session 206 will start the presentation clock.

17 Because the media sinks have already received the initial data of the data  
18 stream, the chances that the media sinks will fall behind (i.e. referred to as a  
19 "glitch") at the beginning of the presentation are greatly reduced if not eliminated  
20 all together. This can effectively provide for a generally seamless presentation  
21 start.

22 At presentation transition boundaries (i.e. when changing from one  
23 presentation to another), media session 206 is configured to attempt to make the  
24 transition seamless, i.e. without interruption between the end of the first  
25 presentation and the beginning of the second. In accordance with this

1 embodiment, the media session 206 accomplishes this by applying some logic to  
2 ensure that the “seamless stream” plays continuously throughout the transition,  
3 without waiting for other streams in the presentation to complete (which may  
4 cause a glitch during the transition).

#### 6 Content Protection

7 In accordance with one embodiment, system 204 and more generally,  
8 systems that employ a media session component as described in this document,  
9 can employ techniques to ensure that media content that is the subject of a  
10 presentation is protected in accordance with rights that may be associated with the  
11 content. This concept is also referred to by some as “digital rights management”.

12 Specifically, certain multimedia content may have specific rights associated  
13 with it. For example, the content provider may wish to restrict playback of this  
14 content to the use of only known, trusted transforms, media sinks and other  
15 components. Accordingly, content protection information associated with the  
16 media content may, but need not then be embedded in the content as will be  
17 appreciated by the skilled artisan. In accordance with this embodiment, media  
18 session 206 is configured to respect any content protection requirements by  
19 validating all of the components that are being inserted into the pipeline and by  
20 making sure that the components are allowed and will be performing allowed  
21 actions on the content. Validation can take place by any suitable measures. For  
22 example, in validating the component, the media session can then validate the  
23 component’s signature, and that the signing authority is a trusted authority.

24 In accordance with one embodiment, the media session 206 can create a  
25 protected media path for such content. The protected media path is configured to

1 make it very difficult if not impossible for unauthorized third parties to intercept  
2 the data flowing through the pipeline.

### 3 4 **Desired Media Engine Configuration**

5 One of the more common scenarios in which the above-described systems  
6 and methods can be employed pertains to setting up a simple playback of a  
7 multimedia presentation. From the application's point of view, it is desirable for  
8 the application to be able to accomplish the following steps in order to configure a  
9 multimedia presentation. The application should be able to create a media engine  
10 and a playback or presentation destination. The application should also be able to  
11 provide a handle to the presentation destination, e.g. a window in which a video  
12 for the presentation should be rendered. The application should also be able to  
13 call IMFMediaEngine::OpenURL, to supply a URL to the multimedia file to be  
14 presented, as well as a pointer to the playback destination. With these capabilities,  
15 the application can now cause the media presentation to be played back by using  
16 the IMFMediaEngine::Start/Stop/Pause APIs. In one embodiment, the application  
17 does not need to wait for any events to arrive as handing of these events are  
18 optional. In another embodiment, the application does handle events from the  
19 media engine for the *open* operation to complete.

### 20 21 **Exemplary Method**

22 Fig. 3 is a method that describes steps in a method in accordance with one  
23 embodiment. The method can be implemented in connection with any suitable  
24 hardware, software, firmware or combination thereof. In one embodiment, the  
25 method can be implemented using the computer architectures such as those

1 described above in connection with Figs. 1 and 2. It is to be appreciated and  
2 understood, however, that other computer architectures can be utilized without  
3 departing from the spirit and scope of the claimed subject matter. In addition, it is  
4 to be appreciated that some of the acts described in this figure need not occur in  
5 the order in which they appear in the flow diagram.

6 Step 300 creates a destination for a presentation. In the example described  
7 above, a destination can be optionally created by an application. Step 302 receives  
8 an *open* call from the application. In the illustrated and described embodiment,  
9 this step is performed by the media engine. Specific examples of calls and  
10 associated call parameters are described above and below. Step 304 ascertains  
11 whether source resolution is needed. Source resolution is the process by which the  
12 media engine causes the appropriate media source (i.e. one that can read the  
13 particular type of data that is to be read) to be created. In the illustrated and  
14 described example, for certain open calls, source resolution is utilized, e.g. for  
15 OpenURL and OpenByteStream calls. In other cases, source resolution is not  
16 utilized because the source is specified, directly or indirectly, in the open call (i.e.  
17 OpenSource, OpenActivate, and OpenTopology). If source resolution is needed,  
18 then step 306 resolves a source to provide a media source and branches to step  
19 308. Examples of how this can be done are given above.

20 If, on the other hand, source resolution is not needed, then the method  
21 branches to step 308 which creates a media session using the media source. This  
22 step can be performed multiple times in the event there are multiple media  
23 sources. Step 310 then ascertains whether the application has specified a topology  
24 that is to be used in presenting the presentation. If no topology is specified by the  
25 application, then step 312 performs a partial topology resolution using either the

1 destination provided by the application or – if no destination was provided – using  
2 an implementation of the destination designed for playback through default  
3 outputs. An example of how this can be performed is described above. Step 314  
4 then performs full topology resolution and branches to step 316. If the application  
5 has specified a topology at step 310, then the method branches to steps 316. Step  
6 316 activates the topology for the presentation and step 318 sends a “media  
7 opened” event to the application. The “media opened” event notifies the  
8 application that it can now control the progress of the presentation by, for  
9 example, calling methods on the media engine.

#### 11 **Distributed (Remote) Presentations**

12 In accordance with one embodiment, the media engine (104, 204) and  
13 media session (106, 206) are also capable of presenting audio and video to devices  
14 existing on remote machines in addition to device that exist on local machines.  
15 One example where this is useful is when an application is using the above  
16 described systems while running under a remote desktop application. One  
17 particular remote desktop application, and one which serves as the basis of the  
18 example described below, is Terminal Services (TS). In Windows XP®, an  
19 exemplary application is the Remote Desktop Connection application. It is to be  
20 appreciated and understood that the concepts described just below can be  
21 employed in connection with different remote applications without departing from  
22 the spirit and scope of the claimed subject matter.

23 In the Terminal Services (TS) case, the user is typically physically sitting at  
24 the console of a TS client, while running the application from a TS server  
25

1 machine. Any media that is to be played, in this example, is sourced from the TS  
2 server, but needs to be sent over a network to the TS client for rendering.

3 In accordance with the described embodiment, the media engine/media  
4 session are configured to send compressed data over the network, in distributed  
5 scenarios, and the work of decompression and rendering is conducted entirely on  
6 the machine where rendering is to take place—i.e. at the TS client. This allows  
7 high-quality content to be played remotely over any particular network  
8 connection. It also ensures that the media content presented to the user goes  
9 through the same components (i.e. decoders and renderers), as is does in the  
10 regular local playback scenario.

11 As an example scenario in which remote presentations can be rendered,  
12 consider Fig. 4 and the system shown generally at 400. System 400 comprises, in  
13 this example, a server 402 and one or more remote clients an exemplary one of  
14 which is shown at 452. Server 402 and remote client 452 are communicatively  
15 linked via a network as indicated. The network can comprise any suitable network  
16 such as, for example, the Internet. In this particular described embodiment, some  
17 of the components are the same as or similar to components discussed above.  
18 Accordingly, for the sake of brevity, some of the components are not described  
19 again.

20 In this example, an application 404 on the server 402 creates a playback  
21 destination 454 in much the same way it does in the local scenario. The  
22 application 402 calls OpenURL on a media engine 406 on server 402 with the  
23 destination 454. The media engine 406 creates a partial topology, as noted above,  
24 and asks the destination 454 for media sinks. The destination 454 then queries  
25 Terminal Services and finds out that the presentation is to take place in connection

1 with a distributed scenario. Information is returned that indicates that the media  
2 sinks are located on a remote machine—the TS client machine 452.

3 The media engine 406 gives its partial topology to a distribution manager  
4 408 to determine whether distribution is needed, and if so, to distribute the  
5 topology between the local and remote machines. For each media stream that  
6 needs to be remoted, the distribution manager performs a number of tasks. First,  
7 the distribution manager 408 creates a network transmitter on the local topology  
8 which will send the compressed media samples over the network to the remote  
9 machine where the samples need to be rendered. As noted, a network transmitter  
10 is created for each stream. Exemplary network transmitters are shown at 410, 412.  
11 It is to be appreciated that the network transmitters are also media sinks, so the  
12 processing pipeline will treat the transmitters as regular sinks and will not need to  
13 do anything special with regard to being in a distributed scenario. Next, the  
14 distribution manager 408 causes network receivers to be placed on the remote  
15 topology. Exemplary network receivers are shown at 456, 458 respectively. The  
16 network receivers receive the compressed media samples from the local  
17 transmitters on the server, and sends them downstream to get rendered by media  
18 sinks 460, 462 respectively. It is to be appreciated and understood that the  
19 network receivers are treated as media sources, so the processing pipeline on the  
20 remote client treats the network receivers as regular media sources without  
21 needing to know that the pipeline is running in a distributed scenario.

22 For each remote machine involved—in this case remote client 452—the  
23 distribution manager 408 creates a media engine 464 on the remote machine and  
24 gives it a presentation destination 454 and the network receivers 456, 458 that it  
25

1 needs to pull data that is to be rendered on its machine. At this point, the  
2 distributed topology is set up, and all machines are ready to start presenting media.

3 The application 404 on the TS server 402 can then call *start* on the media  
4 engine 406, just as it does in the local scenario. The media engine 406 then starts  
5 its local media pipeline and calls *start* on the distribution manager 408 which, in  
6 turn, calls *start* on all media engines (e.g. media engine 464) on the remote  
7 machines. This causes the remote media engines to start their own media  
8 pipelines. Once all media pipelines are running, media flows from the media  
9 source on the TS server 402 through the network transmitters 410, 412 and  
10 receivers 456, 458 to the eventual audio and video renderers on the remote  
11 machines—in this case the TS client 452.

12 It is to be appreciated and understood that the application does not have to  
13 do anything differently between the distributed and local scenarios. The same can  
14 be said of the media source.

### 15 16 **Provision of Additional Services**

17 In accordance with one embodiment, advanced multimedia applications can  
18 take advantage of a wide and extensible array of services made available by the  
19 media engine and media session. These can be accessed by using a  
20 IMFGetService interface exposed by the Media Engine, which will return services  
21 exposed by the media engine, the media session, or any one of the media sources,  
22 sinks, or other components that are in use.

23 This service architecture is extensible and can comprise services such as  
24 rate support and rate control, setting external clocks for reference by the media  
25



1 engine; volume control for audio; frame caching support for editing scenarios, and  
2 video renderer configuration services, to name just a few.

### 3 4 **Application Program Interfaces**

5 The following section provides documentation of APIs associated with an  
6 exemplary implementation of the above-described systems. It is to be appreciated  
7 and understood that APIs other than the ones specifically described below can be  
8 utilized to implement the above-described systems and functionality without  
9 departing from the spirit and scope of the claimed subject matter.

10 The first set of APIs that appear are APIs that are exposed by the media  
11 engine—i.e. IMFMediaEngine APIs.

#### 12 13 **IMFMediaEngine**

14 {  
15 // Open Methods  
16 HRESULT OpenURL( [in] LPCWSTR pwszURL,  
17 [in] IUnknown \*pStreamSelection,  
18 [in] IMFDestination \*pDestination );  
19 HRESULT OpenSource( [in] IMFMediaSource \*pSource,  
20 [in] IUnknown \*pStreamSelection,  
21 [in] IMFDestination \*pDestination );  
22 HRESULT OpenActivate( [in] IActivate \*pActivate,  
23 [in] IUnknown \*pStreamSelection,  
24 [in] IMFDestination \*pDestination );  
25 HRESULT OpenByteStream( [in] LPCWSTR pwszURL,  
[in] IMFByteStream \*pByteStream,  
[in] IUnknown \*pStreamSelection,  
[in] IMFDestination \*pDestination );  
HRESULT OpenTopology( [in] IMFTopology \*pTopology,  
[in] IUnknown \*pStreamSelection,  
[in] IMFDestination \*pDestination );  
HRESULT Close();

```

1      HRESULT Shutdown();
2
3      // GetState, GetCapabilities
4      HRESULT GetState( [out] IMFMediaEngineState **ppState );
5      HRESULT GetCapabilities( [out] DWORD *pdwCaps );
6
7      // Start/Stop/Pause
8      HRESULT Start( [in] const GUID *pguidTimeFormat,
9                    [in] const PROPVARIANT *pvarStartPosition,
10                   [in] const PROPVARIANT *pvarEndPosition );
11      HRESULT Stop();
12      HRESULT Pause();
13
14      // GetDestination
15      HRESULT GetDestination( [out] IMFDestination **ppDestination );
16
17      // GetClock
18      HRESULT GetClock( [out] IMFClock** ppClock );
19
20      // GetMetadata
21      HRESULT GetMetadata( [in] REFIID riid,
22                          [out] IUnknown** ppunkObject );
23
24      // Statistics
25      HRESULT GetStatistics( [out] IUnknown **ppStats );
26
27      };

```

### IMFMediaEngine::OpenURL()

The OpenURL provides Open functionality in the case where the data source is identified through a URL (Uniform Resource Locator). The media engine will resolve this URL to the appropriate source object for use in the engine.

#### **Syntax**

```

      HRESULT OpenURL( [in] LPCWSTR pwszURL,
                      [in] IUnknown *pStreamSelection,
                      [in] IMFDestination *pDestination );

```

#### **Parameters**

*pwszURL*

[out] Specifies the Uniform Resource Locator to be opened.

*pStreamSelection*

[in] Specifies the stream and format selection parameters to be used.

*pDestination*

[in] Specifies the destination for the data.

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### Remarks

This is an asynchronous call; if it returns successfully, an MEMediaOpened event will be generated asynchronously. If the process fails asynchronously, this event will contain a failure code.

### IMFMediaEngine::OpenSource()

The OpenSource method is similar to the OpenURL except it accepts a Media Source object as input in place of a URL. The media engine bypasses the source resolution step and performs the same remaining steps as described in the Open documentation.

### Syntax

```
HRESULT OpenSource( [in] IMFMediaSource *pSource,  
                    [in] IUnknown *pStreamSelection,  
                    [in] IMFDestination *pDestination );
```

### Parameters

*pSource*

[out] Specifies a pointer to the Media Source to provide data.

*pStreamSelection*

[in] Specifies the stream and format selection parameters to be used.

*pDestination*

[in] Specifies the destination for the data.

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### Remarks

1 This is an asynchronous call; if it returns successfully, an MEMediaOpened  
2 event will be generated asynchronously. If the process fails asynchronously, this  
3 event will contain a failure code.

#### 4 IMFMediaEngine::OpenActivate

5 The OpenActivate method provides Open functionality, where the data  
6 source is provided as an IActivate. Typically this IActivate is a retrievable  
7 parameter and represents some sort of hardware device capable of sourcing media  
8 data.

#### 9 **Syntax**

10 HRESULT OpenActivate( [in] IActivate \*pActivate,  
11 [in] IUnknown \*pStreamSelection,  
[in] IMFDestination \*pDestination );

#### 12 **Parameters**

*pActivate*

13 [out] Specifies a pointer to the Function Instance to provide data.

*pStreamSelection*

14 [in] Specifies the stream and format selection parameters to be used.

*pDestination*

15 [in] Specifies the destination for the data.

#### 16 **Return Values**

17 If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

#### 18 **Remarks**

19 This is an asynchronous call; if it returns successfully, an MEMediaOpened  
20 event will be generated asynchronously. If the process fails asynchronously, this  
21 event will contain a failure code.

#### 22 IMFMediaEngine::OpenByteStream()

23 The OpenByteStream method is similar to the OpenURL except it accepts a  
24 Byte Stream object as input in place of a URL. The media engine bypasses the  
25

1 scheme resolution step and performs the same remaining steps as described in the  
2 Open section.

### 3 **Syntax**

4 HRESULT OpenByteStream( [in] LPCWSTR pwszURL,  
5 [in] IMFByteStream \*pByteStream,  
6 [in] IUnknown \*pStreamSelection,  
7 [in] IMFDestination \*pDestination );

### 8 **Parameters**

9 *pwszURL*

10 [in] Optionally specifies the URL of the provided Byte Stream. May be  
11 NULL if the URL is unknown.

12 *pByteStream*

13 [in] Specifies a pointer to Byte Stream object to provide data.

14 *pStreamSelection*

15 [in] Specifies the stream and format selection parameters to be used.

16 *pDestination*

17 [in] Specifies the destination for the data.

### 18 **Return Values**

19 If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### 20 **Remarks**

21 This is an asynchronous call; if it returns successfully, an MEMediaOpened  
22 event will be generated asynchronously. If the process fails asynchronously, this  
23 event will contain a failure code.

24 The pwszURL argument is optional; if the URL of the Byte Stream is  
25 known it should be provided to allow efficient source resolution.

### 26 IMFMediaEngine::OpenTopology()

27 The OpenTopology method is similar to the OpenURL except it accepts a  
28 Topology object as input in place of a URL. The media engine bypasses normal  
29 pipeline construction logic and simply uses the Topology provided.

### 30 **Syntax**

31 HRESULT OpenTopology( [in] IMFTopology \*pTopology,  
32 [in] IUnknown \*pStreamSelection,

[in] IMFDestination \*pDestination );

### Parameters

*pTopology*

[out] Specifies a pointer to Byte Stream object to provide data.

*pStreamSelection*

[in] Specifies the stream and format selection parameters to be used.

*pDestination*

[in] Specifies the destination for the data.

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### Remarks

This is an asynchronous call; if it returns successfully, an MEMediaOpened event will be generated asynchronously. If the process fails asynchronously, this event will contain a failure code.

NULL may be passed for the pDestination parameter in this case, indicating that the topology contains all necessary information regarding the media sinks. In this case a source-initiated topology change will be treated as a failure case.

### IMFMediaEngine::Close()

The Close method provides a way to end processing media samples and close the currently active media source.

### Syntax

HRESULT Close();

### Parameters

*None*

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### Remarks

This is an asynchronous method; if it succeeds, an MEMediaClosed event is generated. This event may contain a failure code in the case that the Media Engine fails asynchronously.

## IMFMediaEngine::Shutdown()

The Shutdown method causes all the resources used by the Media Engine to be properly shutdown and released. This is an asynchronous operation which queues the MEShutdown event upon completion. Upon successful completion of this operation all methods on the Media Engine will return MF\_E\_SHUTDOWN when called.

### **Syntax**

HRESULT Shutdown();

### **Parameters**

*none*

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### **Remarks**

This method asynchronously queues an MEShutdown event. All referenced media engine must still be released (via the IUnknown::Release method) to ensure no memory is leaked. Upon successful completion of this operation, all methods (including calling Shutdown again) on the media engine will return MF\_E\_SHUTDOWN. The exception is the IMFMediaEventGenerator calls which are still allowed. However, the Media Engine will queue no more events to the event generator after queuing the MEShutdown event. After calling Shutdown, the user can call the global MF function MFShutdown instead of waiting for the MEShutdown event. MFShutdown will ensure that the Shutdown operation runs to completion. Note that events may be queued and callbacks invoked in the process of calling MFShutdown.

## IMFMediaEngine::GetState

The GetState method synchronously fetches the current state of the engine.

### **Syntax**

HRESULT GetState( [out] IMFMediaEngineState \*\*ppState );

## Parameters

*ppState*

[out] A pointer to an IMFMediaEngine interface. The methods of the IMFMediaEngineState interface give more information about the state.

## Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## IMFMediaEngine::GetCapabilities

GetCapabilities returns a DWORD describing what operations are allowed on the engine at the current moment.

## Syntax

```
HRESULT GetCapabilities( DWORD *pdwCaps );
```

## Parameters

*pdwCaps*

[out] A bitwise OR of the current capabilities of the engine.

## Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## IMFMediaEngine::Start()

The Start method provides an asynchronous way to start processing media samples. This operation gets the media samples moving through the pipeline and delivered to the appropriate sinks. The *StartOffset* parameter lets you specify the location in the media at which to start processing. The *Duration* parameter lets you specify the length of media that needs to be processed beginning at the *StartOffset*. Alternately or additionally, an *EndOffset* parameter can be used to specify the location in the media at which to end processing. An MEMediaStarted event is generated to mark the completion of this asynchronous operation.



### Syntax

```
HRESULT Start( [in] const GUID *pguidTimeFormat,  
               [in] const PROPVARIANT *pvarStartPosition,  
               [in] const PROPVARIANT *pvarEndPosition );
```

### Parameters

**guidTimeFormat**  
[in] Specifies the time format used for pvarStartPosition and pvarEndPosition parameters.

**pvarStartPosition**  
[in] A PROPVARIANT specifying the start time, in the units specified by guidTimeFormat, at which to start processing

**pvarEndPosition**  
[in] A PROPVARIANT specifying the end time, in the units specified by guidTimeFormat, at which to end processing

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### Remarks

Use GUID\_NULL for guidTimeFormat to indicate that llStartOffset is in the standard (100-ns) units.

### IMFMediaEngine::Stop()

The Stop method provides an asynchronous way to stop media sample processing in the engine. An MEMediaStopped event is generated to mark the completion of this operation.

### Syntax

```
HRESULT Stop();
```

### Parameters

none

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### IMFMediaEngine::Pause()

1       The Pause method provides an asynchronous way to stop media sample  
2 processing in the engine. An MEMediaPaused event is generated to mark the  
3 completion of this operation.

4               **Syntax**

5               HRESULT Pause();

6               **Parameters**

7               none

8               **Return Values**

9               If the method succeeds, it returns S\_OK. If it fails, it returns an error  
10 code.

11       IMFMediaEngine::GetDestination()

12       The GetDestination method returns the current destination, as specified in  
13 one of the OpenXXX methods.

14               **Syntax**

15               HRESULT     GetDestination(     [out]     IMFDestination  
16               \*\*ppDestination );

17               **Parameters**

18               ppDestination  
19               [out] Specifies a pointer to a variable where the destination will be  
20 stored.

21               **Return Values**

22               If the method succeeds returns S\_OK. If it fails, a suitable error code  
23 is returned.

24       IMFMediaEngine::GetClock()

25       The GetClock method returns the presentation clock being used for  
rendering the current media.

### Syntax

```
HRESULT GetClock( [out] IMFClock** ppClock );
```

### Parameters

ppClock

[out] Specifies a pointer to a variable where the clock object will be stored.

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### IMFMediaEngine::GetMetadata()

The GetMetadata method provides a way to access the metadata for the current media.

### Syntax

```
HRESULT GetMetadata( [in] REFIID riid,  
[out] IUnknown** ppunkObject );
```

### Parameters

riid

[in] Specifies the IID of the interface being requested.

ppunkObject

[out] Specifies a pointer to a variable where the requested object will be stored.

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### Remarks

The riid parameter is not supported currently. A property store object is returned in all cases.

### IMFMediaEngine::GetStatistics()

1 The GetStatistics method provides a way to access the statistics.

2  
3 **Syntax**

HRESULT GetStatistics( [out] IUnknown \*\*ppStats );

4 **Parameters**

5 ppStats

[in] The stats.

6  
7 **Return Values**

8 If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

9  
10 **IMFStreamSelector**

11 The media engine exposes the IMFStreamSelector interface as part of the  
12 Stream Selector Service mentioned above.

13  
14 interface IMFStreamSelector : IUnknown

{

15 HRESULT SetManualStreamSelection( [in] BOOL fManual );

HRESULT GetManualStreamSelection( [out] BOOL \*pfManual );

16  
17 HRESULT SetAutomaticStreamSelectionCriteria( [in] IPropertyStore  
18 \*pCriteria );  
};

19  
20 IMFStreamSelector::SetManualStreamSelection()

21 The SetManualStreamSelection method configures the stream selection  
22 mode of the Media Engine.

23 **Syntax**

24 HRESULT SetManualStreamSelection( [in] BOOL fManual );

25 **Parameters**

*fManual*

[in] FALSE for automatic stream selection, any other value for manual stream selection.

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### **Remarks**

This method may only be called when the Media Engine is in the stInitial or stOpened states.

### IMFStreamSelector::GetManualStreamSelection()

The GetManualStreamSelection method retrieves the stream selection mode of the media engine.

### **Syntax**

HRESULT SetManualStreamSelection( [in] BOOL \*pfManual );

### **Parameters**

pfManual

[out] A pointer to a BOOL that retrieves the mode.

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### IMFStreamSelector::SetAutomaticStreamSelectionCriteria()

The SetAutomaticStreamSelectionCriteria method sets the stream selection criteria that the Media Engine uses.

### **Syntax**

HRESULT SetAutomaticStreamSelectionCriteria( [in]  
IPropertyStore \*pCriteria );

### **Parameters**

pCriteria

[out] The new criteria to use.

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### **Remarks**

This method may only be called if the Media Engine is in automatic stream selection mode. Otherwise an error will be returned. Any of the OpenXXX() methods on the Media Engine also set the stream selection criteria.

### **States**

In the illustrated and described embodiment, the media engine exposes a state. This allows applications to query the engine synchronously and get a reasonable idea of what the engine is doing without requiring the application to listen to the history of events up until the current time.

In the illustrated and described embodiment, the engine states are fairly broad and varied. For more granular information, the application must listen to events or use services off the engine. Information that is available via services is in general not duplicated in the state model. The data type used to express state is a structure, with a major type and additional state specific attributes. The simplest callers may choose to look only at the major type. Advanced callers may want to look at the state specific attributes.

### **EngineStateType enum**

The major types of the states are values from the EngineStateType enum. In this example, there are eight states. The first six can be considered primary states, and the last two can be considered secondary states.

enum EngineStateType

```
{  
    StInitial,  
    StConnecting,  
    StOpened,  
    StRunning,  
    StPaused,  
    StShutdown,  
    StTransitioning,  
    StSuspended  
};
```

### StInitial

When the engine is just created it is in this state. Subsequently when IMFMediaEngine::Close is called the engine returns to this state after the close action completes.

### StConnecting

When the engine is trying to open a networked media and is connecting to the server, the engine goes into this state. While the engine is running if the connection is lost and the engine attempts to reconnect, the engine goes into this state again.

### StOpened

When the engine has successfully opened a media, via OpenURL, OpenSource, OpenActivate, OpenByteStream, or OpenTopology, the engine goes into this state. Subsequently the engine returns to this state if playback stops either because the client called IMFMediaEngine::Stop, or because the end of the media was reached.

1  
2       StRunning

3       When the engine has successfully started as a result of  
4       IMFMediaEngine::Start, the engine goes into this state. It does not matter what  
5       rate the engine is running at.

6  
7       StPaused

8       When the engine has successfully paused as a result of  
9       IMFMediaEngine::Pause, the engine goes into this state.

10  
11       StShutdown

12       After the engine has shutdown as a result of IMFMediaEngine::Shutdown,  
13       the engine goes into this state.

14  
15       StTransitioning

16       This is a secondary state. When an asynchronous call is made on the  
17       engine, the engine immediately goes into this state. When the asynchronous call  
18       completes, the engine goes into the target state. The EngineState structure contains  
19       two additional parameters if the major type is StTransitioning. These parameters  
20       indicate the major type of the previous state, and an EngineMethodID enum  
21       describing the API call being processed currently.

22  
23       StSuspended

24       This is also a secondary state. If the opening or running of the engine is  
25       waiting on some user action then the engine goes into this state. The EngineState



1 structure contains one additional parameter in this case. The additional parameter  
2 is an EngineSuspensionReason enum indicating why the engine was suspended.

### 4 **EngineMethodID**

5 This is an enum that lists all methods of the engine that could result in a  
6 state change. Most of these are methods on the IMFMediaEngine interface with  
7 the exception of eapiRateChange.

```
8
9     enum EngineMethodID
10    {
11        eapiOpenURL,
12        eapiOpenSource,
13        eapiOpenActivate,
14        eapiOpenByteStream,
15        eapiOpenTopology,
16        eapiStart,
17        eapiPause,
18        eapiStop,
19        eapiClose,
20        eapiShutdown,
21        eapiRateChange // User is trying to change rate via the rate control
22    };
23
24 service
25     };
```

### 18 **EngineSuspensionReason**

19 This is a list of reasons why the engine is suspended waiting on user action.

```
20
21     enum EngineSuspensionReason
22    {
23        esuspLicenseAcquisition,
24        esuspIndividualization,
25        esuspCodecMissing
26    };
27
```

## **IMFMediaEngineState interface**

From the description of states so far it may be evident that a state comprises a type and some additional attributes. All this is packaged into an IMFMediaEngineState interface. Packaging this as an interface allows for passing the state around in a compact manner. It also gives an option of adding more state types and more state-specific attributes in the future. This interface has two methods as described below.

### **IMFMediaEngineState::GetStateType()**

This method fetches the major type of the state. E.g. stRunning, stPaused.

#### **Syntax**

HRESULT IMFMediaEngineState::GetStateType( [out] EngineStateType \*pStateType)

#### **Parameters**

pStateType

[out] A pointer to an EngineStateType enum. This is filled with the major type (one of the enum values of EngineStateType).

#### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### **IMFMediaEngineState::GetAttribute()**

This method fetches additional attributes of the state. The available attributes are specific to the major type of the state e.g. if the major type of the state is stTransitioning, then an attribute MF\_ENGINE\_METHOD\_ID will be available, whose value is one of the EngineMethodID enum values.

1 This attribute will not be available if the major type of the state is anything  
2 else.

### 3 **Syntax**

4 HRESULT IMFMediaEngineState::GetAttribute( [in] GUID  
5 guidAttribute, [out] PROPVARIANT \*pvarAttribute )

### 6 **Parameters**

7 guidAttribute

[in] The guid of the attribute whose value is desired.

8 pvarAttribute

9 [out] The value of the attribute.

### 10 **Return Values**

11 If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## 12 **Media Session Application Program Interfaces**

13 The following discussion provides details with respect to APIs that are  
14 exposed by a media session.  
15

### 16 **IMFMediaSession**

#### 17 **IMFMediaSession::SetTopology()**

18 The SetTopology method provides an asynchronous way of initializing a  
19 full topology on the media session. The ResolveTopology method can be used to  
20 resolve a partial one into a full topology. An MENewPresentation event is  
21 generated to mark the completion of the SetTopology call.  
22  
23  
24

### 25 **Syntax**

```
1      HRESULT SetTopology(  
2          [in] IMFTopology* pTopology  
3          );
```

### Parameters

*pTopology*

[out] Specifies a pointer to a full topology.

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### IMFMediaSession::Start()

The Start method provides an asynchronous way to start processing media samples. This operation gets the media samples moving through the pipeline and delivered to the appropriate sinks. The *StartOffset* parameter lets you specify the location in the media at which to start processing. The *Duration* parameter lets you specify the length of media that needs to be processed beginning at the *StartOffset*. An *MESessionStarted* event is generated to mark the completion of this asynchronous operation.

### Syntax

```
16      HRESULT Start([in] const GUID *pguidTimeFormat,  
17                    [in] const PROPVARIANT *pvarStartPosition,  
18                    [in] const PROPVARIANT *pvarEndPosition    );
```

### Parameters

*guidTimeFormat*

[in] Specifies the time format used for *pvarStartPosition* and *pvarEndPosition* parameters.

*pvarStartPosition*

[in] A PROPVARIANT specifying the start time, in the units specified by *guidTimeFormat*, at which to start processing

*pvarEndPosition*

[in] A PROPVARIANT specifying the end time, in the units specified by *guidTimeFormat*, at which to end processing

### Return Values

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

#### **Remarks**

Use GUID\_NULL for guidTimeFormat to indicate that llStartOffset is in the standard (100-ns) units.

#### **IMFMediaSession::Pause()**

The Pause method provides an asynchronous way to stop media sample processing in the session. An MESessionPaused event is generated to mark the completion of this operation.

#### **Syntax**

HRESULT Pause();

#### **Parameters**

None

#### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

#### **IMFMediaSession::Stop()**

The Stop method provides an asynchronous way to stop media sample processing in the session. An MESessionStopped event is generated to mark the completion of this operation.

#### **Syntax**

HRESULT Stop();

#### **Parameters**

none

#### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## IMFMediaSession::Shutdown()

The Shutdown method causes all the resources used by the media session to be properly shutdown and released. This is a synchronous operation and upon successful completion of this operation all methods on the Media Session will return failures if called.

### **Syntax**

HRESULT Shutdown();

### **Parameters**

none

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## IMFMediaSession::SetPresentationClock()

The SetPresentationClock method provides a way to specify the presentation clock to be used in rendering the current media session.

### **Syntax**

HRESULT SetPresentationClock(  
    [in] IMFPresentationClock\* pClock  
);

### **Parameters**

*ppClock*

[in] Specifies a pointer to a presentation clock object.

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

## IMFMediaSession::GetPresentationClock()

The GetPresentationClock method returns the presentation clock being used for rendering the current media session.

### **Syntax**

```
HRESULT GetPresentationClock(  
    [out] IMFPresentationClock** ppClock  
);
```

### **Parameters**

*ppClock*

[out] Specifies a pointer to a variable where the presentation clock object will be stored.

### **Return Values**

If the method succeeds, it returns S\_OK. If it fails, it returns an error code.

### **Conclusion**

The media processing methods, systems and application program interfaces (APIs) described above provide a simple and unified way of rendering media from an origin to a destination of choice without requiring intimate knowledge about the underlying components, their connectivity and management. The systems and methods are flexible in terms of their implementation and the various environments in which the systems and methods can be employed. The systems, methods and APIs can be employed by applications that range from simple to complex, thus further extending the various scenarios in which the systems, methods and APIs can be employed.

Although the invention has been described in language specific to structural features and/or methodological steps, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or

1 steps described. Rather, the specific features and steps are disclosed as preferred  
2 forms of implementing the claimed invention.  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25